基于FPGA的高精度科学计算加速器研究

雷元武 窦 勇 郭 松
(国防科学技术大学计算机学院 长沙 410073)

摘 要 探索了FPGA平台加速高精度科学计算应用的能力和灵活性.首先,研究科学计算中最常用的操作——向量内积,提出基于定点操作的精确向量内积算法.以IEEE 754-2008标准的四精度(Quadruple Precision)浮点算术为例,在FPGA平台上设计了一个基于全展开方法的全流水四精度浮点乘累加单元(QPMAC):提出两级存储策略精确存储乘累加和;采用保留进位累加策略减少定点加法器位宽、简化进位处理、优化关键路径;引入累加和划分策略,实现流水吞吐率.最后,在XC5VLX330 FPGA芯片上设计一个LU分解和MGS-QR分解加速器原型来验证QPMAC的性能.实验结果表明,与运行在Intel四核处理器上的基于OpenMP的并行算法相比,集成4个QPMAC单元的加速器能获得42倍到97倍的性能提升,并且能获得更高结果精度和更低能量消耗.

关键词 四精度浮点算术;LU分解;MGS-QR分解;FPGA;硬件加速器;E量级计算
中图法分类号 TP302 **DOI号**: 10.3724/SP.J.1016.2012.00112

High Precision Scientific Computation Accumulator on FPGA

LEI Yuan-Wu DOU Yong GUO Song
(*College of Computer, National University of Defense Technology, Changsha* 410073)

Abstract In this paper we explore the capability and flexibility of FPGA solutions in a sense to accelerate high precision scientific computing applications. First, we research the inner product operation, which occurs in almost all scientific and engineering applications, and propose the exact inner product algorithm based on exact long fixed-point operations. Taking IEEE 754-2008 quadruple precision floating-point as an example, we have implemented a full-pipelined Quadruple Precision Multiplication and Accumulation (QPMAC) into FPGA devices. We propose a two-level RAM banks scheme to store the exact fixed-point result, and use carry-saved accumulator scheme to minimize the width of fixed-point adder and simplify the logic of carry resolution. We also introduce a scheme of partial summation to enhance the pipeline throughput of MAC operations, by dividing the summation function into 4 partial operations, processed in 4 banks. To prove the concept, we prototype four QPMAC units into a XC5VLX330 FPGA chip and perform LU decomposition and MGS-QR decomposition. The experimental results show that our implementations based on FPGA achieve 42X-97X better performance, more precision results and much lower power consumption compared with the use of a parallel software approach based on OpenMP running on an Intel Core2 Quad Q8200 CPU at 2.33 GHz.

Keywords quadruple precision floating-point arithmetic; LU decomposition; MGS-QR decomposition; FPGA; hardware accelerator; ExeScale computation

收稿日期:2011-02-22;最终修改稿收到日期:2011-05-13.本课题得到国家"八六三"高技术研究发展计划项目基金(2008AA01A201)和国家自然科学基金重点项目(60833004,61125201)资助.**雷元武**,男,1982年生,博士研究生,主要研究方向为高性能计算机体系结构.E-mail:yuanwulei@nudt.edu.cn.窦 勇,男,1966年生,博士,研究员,博士生导师,主要研究领域为计算机体系结构、高性能嵌入式微处理器、可重构计算.**郭 松**,男,1985年生,博士研究生,主要研究方向为计算机体系结构.

1 引　言

大部分科学计算采用双精度浮点算术实现. 由于浮点数据是对现实世界中精确数据的近似表示，运算过程中存在舍入误差，这种舍入误差的累积将导致计算结果不精确、不可靠、甚至不正确. 预测到 E 量级(ExaScale：百万万亿次/秒)[1]计算时代，LU 分解的计算结果仅有 3 位有效[2]，这表明双精度浮点算术不能满足未来科学计算的精度要求.

高精度计算是解决精度问题最直接、有效、可靠的方法，已经广泛应用于各个领域，如天气或气候模拟、超新星模拟、计算几何、数值理论等[3]. 鉴于高精度计算在提高应用结果精度、数值算法稳定性和结果再现性等方面的优势，IEEE 754-2008 浮点算术标准中增加了四精度(Quadruple Precision)浮点格式[4]来支持高精度计算. 目前，四精度浮点算术通常使用软件模拟实现，如 Intel Fortran[5]、GMP、MPFR[6]、QD(Quad-Double)[7]等函数库. 软件方法的最大缺点是计算性能低，相对于双精度浮点算术，四精度浮点算术的性能降低了一个数量级，Linpack 测试时间为双精度浮点算术的 36 倍[3,8].

有学者设计专用硬件逻辑来支持四精度浮点算术，克服软件方法的性能障碍. IBM S/390 G5 处理器[9]在硬件上支持四精度浮点基本操作. Akkas 等人[10]设计了双模式的四精度加法、乘法和除法单元，支持一个四精度浮点操作或者两个并行双精度浮点操作.

使用可重构平台加速科学应用已经成为一种主流趋势，在高精度科学应用中具有较高的性能和良好的可扩展性[2]. 可重构计算通过定制逻辑满足应用对计算精度的需求，以达到最佳性能. 可重构计算的底层技术是 FPGA(现场可编程门阵列)技术. 目前，FPGA 向更大密度、更高性能和更低功耗发展，Xilinx Virtex-6 FPGA 具有 741K 个逻辑单元，37.4Mbit 嵌入式存储器和 2016 个 DSP48E 乘法器. 文献[2]在 FPGA 平台上设计了 Double-Double (DD)和 QD 类型的科学应用加速器，相对于通用 CPU，能够取得 30 倍以上的加速比.

本文在文献[2]的基础上，以 IEEE 754-2008 标准的四精度浮点算术为例，在 FPGA 平台上设计了一个基于全展开方法的全流水四精度浮点乘累加单元(QPMAC)，采用无误差的定点操作代替浮点操作以获得精确的结果，并提出两级存储策略、保留进位累加策略、累加和划分策略，来提高 QPMAC 性能，实现流水吞吐率. 最后在 FPGA 芯片上设计 LU 分解和 MGS-QR(Modified Gram-Schmidt QR)分解加速器原型来验证 QPMAC 的正确性及性能.

2 背景介绍

2.1 浮点算术

一个浮点算术系统可以用四元组 $R(b,r,e_1,e_2)$ 表示，其中 b 表示基(2 或 10)；r 表示尾数的位数，即精度；e_1 为最小指数，e_2 为最大指数. 表 1 列举了部分浮点算术系统.

表 1　几种浮点算术系统定义

格式	定义	位数	精度
四精度[4]	$R\{2,113,-16382,16383\}$	128	$\approx 10^{-34}$
DD[7]	$\approx R\{2,106,-1022,1023\}$	128	$\approx 10^{-32}$
QD[7]	$\approx R\{2,212,-1022,1023\}$	256	$\approx 10^{-64}$

QD 函数库采用多个标准的浮点数据和来表示一个高精度数据. 这类高精度算术充分利用当今处理器提供的高性能浮点处理能力，计算速度较快，但是这种实现方式的精度是确定的，数据的表示范围较小，仅为双精度浮点表示范围.

2.2 浮点算术中的精度损失

浮点算术的精度损失来源于截断误差和基本操作的舍入误差. 截断误差是将精确值转换为确定格式浮点数据时产生的误差. 基本浮点算术操作中，减法对精度损失影响最大，两个非常相近的数据之间的减法操作存在“巨量消失”现象[11]，计算结果的相对误差变大，不精确位向前传播多位.

文献[2]以 LU 分解为例，通过大量随机测试，建立精度损失与基本操作数量的关系模型. LU 分解结果的不精确位数(m)是以矩阵规模(n)自然对数的方式增加($m=3.11\times\ln(n)-1.499$)，预测在即将到来的 E 量级计算时代，结果的平均精度仅有 3 位有效(十进制). 这不能满足科学计算的精度要求，因此，我们必须构建更高精度的算术单元.

2.3 高精度向量内积的相关研究

向量内积($\sum_{i=1}^{n} x_i \times y_i$)是最重要的基本操作之一，也是 BLAS 函数库中的一个基本子程序，几乎在所有科学和工程应用中出现. 在矩阵类运算、Krylov 子空间等应用中，对向量内积的计算精度要求较高[12]. 但是，通用处理器没有专用的硬件来实现向量内积，只能通过基本浮点操作模拟实现. 而基

本浮点操作的舍入误差可能导致内积结果不精确.为了提高计算精度,Higham[13]引入了重排序方法和补偿累加方法.重排序方法首先按升序或降序对累加数据进行排序,然后对有序数组进行累加,这种方法增加排序开销;补偿累加方法记录和累加每次舍入的信息,最后将其补偿到最终结果中,这种方法需要额外开销完成舍入信息的累加.Rump 等人[14]提出了精确累加算法,其关键是使用一种无误差划分技术.以上方法都是使用软件方法来提高计算精度,不能有效避免累加过程中的"巨量消失"现象,而且软件方法的速度较慢.

还有学者提出高精度向量内积单元的设计方法.Kulisch[15]建议将精确内积单元集成到通用处理器中,作为"第 5 个基本浮点算术".He 等人[16]提出了累加数据分组方法,每组数据移位对齐后再进行定点累加,这种方法在每组计算中需要移位对齐、规格化处理,增加处理开销.Kulisch[17]和 Knofel[18]设计了一个标量内积单元(SPU)协处理器实现 32 位和 64 位浮点精确向量内积.

我们设计了 DD 类型和 QD 类型的高精度乘累加单元(HPMAC)[2].从表 1 可知,相对于 DD 和 QD 类型,IEEE 754-2008 标准的四精度浮点格式的数据范围扩大了 16 倍,这意味着 QPMAC 设计中用于快速进位处理的标志位宽度也需要扩展 16 倍.这将给全流水 QPMAC 设计带来挑战——如何快速定位进位终止因子(Carry Terminate Factor)和确定进位传递因子(Carry Skip Factor).本文在文献[2]的基础上对基于全展开的全流水高精度乘累加器的设计及 FPGA 加速高精度科学应用的探讨进行完善:首先,采用保留进位累加策略来替换快速进位策略,以简化进位处理,支持更大数据范围的浮点格式,并以 IEEE 754-2008 标准的四精度浮点为例进行验证;其次,增加 MGS-QR 分解应用来说明 QPMAC 的性能及 FPGA 加速高精度科学应用的能力;最后,使用性能较高的任意精度函数库(MPFR)和针对 Intel 处理器进行性能优化的 Intel Fortran 函数库来对比基于 QPMAC 单元的 FPGA 加速器的性能.为了公平对比多核 CPU 和 FPGA 平台的性能,我们采用 OpenMP 技术对 LU 分解和 MGS-QR 分解应用进行并行化,充分开发多核 CPU 的性能.

3 高精度浮点乘累加单元的 FPGA 实现

高精度浮点乘累加操作首先进行无精度损失的定点乘法和加法得到精确定点乘累加和,然后对其进行规格化.该操作仅在规格化过程中进行一次舍入操作,最多引入一位误差.

3.1 精确向量内积算法

假定 $sign(x)$(或 SX),$exp(x)$(或 EX)和 $mant(x)$(或 MX)分别表示浮点数据 x 的符号、指数和尾数;$\boldsymbol{A}=(A_i)$和 $\boldsymbol{B}=(B_i)(i=1,2,\cdots,n)$表示两个长度为 n 的向量,其中 $x,A_i,B_i\in(2,r,e_1,e_2)$.向量内积为 $\boldsymbol{s}=\sum_{i=1}^{n}A_i\cdot B_i$.乘积 $P=A_i\cdot B_i$的尾数宽度为 $2r$ 位,指数范围为$[2e_1,2e_2]$,即 $P\in(2,2r,2e_1,2e_2)$.因此,乘积 P 所有位的信息可以无损失地存储到一个长度为 $L=|2e_1|+2r+2e_2$的寄存器中.

图 1 描述了精确向量内积算法.首先,读取浮点数据 A_i和 B_i,并将其分解为符号位、指数位、尾数位;然后,执行定点尾数乘法得到宽度 $M=2r$ 的乘积 *product*;再将 *product* 累加到寄存器 *sum* 中;最后,规格化 L 位的累加和 *sum* 为标准格式数据.

```
Algorithm : Exact Dot Product
Input: A_i, B_i(i=1,…,n)
Output: Result
Initialize: sum[L:1]=0;
 1. for i=1 to n do
 2.   Load(A_i, B_i);
 3.   s=sign(A_i)^sign(B_i); exp=exp(A_i)+exp(B_i);
        product=mant(A_i) * mant(B_i);
 4.   new_sum[2^exp-1:1]=sum[2^exp-1:1];
 5.   if (s==0) then
 6.     {carry, new_sum[2^exp+M-1:2^exp]}=sum[2^exp+M-1:
          2^exp]+product[M:1];
 7.     new_sum[L:2^exp+M]=sum[L:2^exp+M]+carry;
 8.   else
 9.     {carry, new_sum[2^exp+M-1:2^exp]}=sum[2^exp+M-1:
          2^exp] -product[M:1];
10.     new_sum[L: 2^exp+M]=sum[L:2^exp+M] -carry;
11.   end if
12.   sum[L:1]=new_sum[L:1];
13. end for
14. Result=normalize(sum[L:1])
```

图 1 精确向量内积算法

算法的步 4~11 将尾数乘积 *product* 累加到 L 位累加和 *sum* 中,通常,需要一个 L 位的定点加法器,这会造成较大的延时.然而,我们可以根据乘积指数 exp,将 *sum* 分为 3 个部分,定点累加过程只需一个 M 位加法器.累加和 *sum* 的小数点位于$|2e_1|+2r$,*product* 对应于 *sum* 的 2^{exp}到 $2^{exp}+M-1$ 的位置.累加和 *sum* 低位部分保持不变(算法的第 4 行).仅需要一个 M 位的定点加法完成 *product* 对应位置的累加(算法的第 6 行或第 9 行).高位部分的操作取决于 M 位加法的进位信息(算法的第 7 行或第 10 行).

3.2　四精度浮点乘累加单元设计

如图 2 左边所示，四精度浮点乘累加单元(QPMAC)主要由 113×113 定点尾数乘法器，尾数乘积累加模块和规格化模块组成.

使用 Xilinx Virtex-5 系列 FPGA 芯片中的 DSP48E(18×25)单元来构建 113×113 定点尾数乘法模块. 该模块采用 6 级流水结构，包括 35 个 DSP48E 单元和部分积加法树，乘积结果为 226 位.

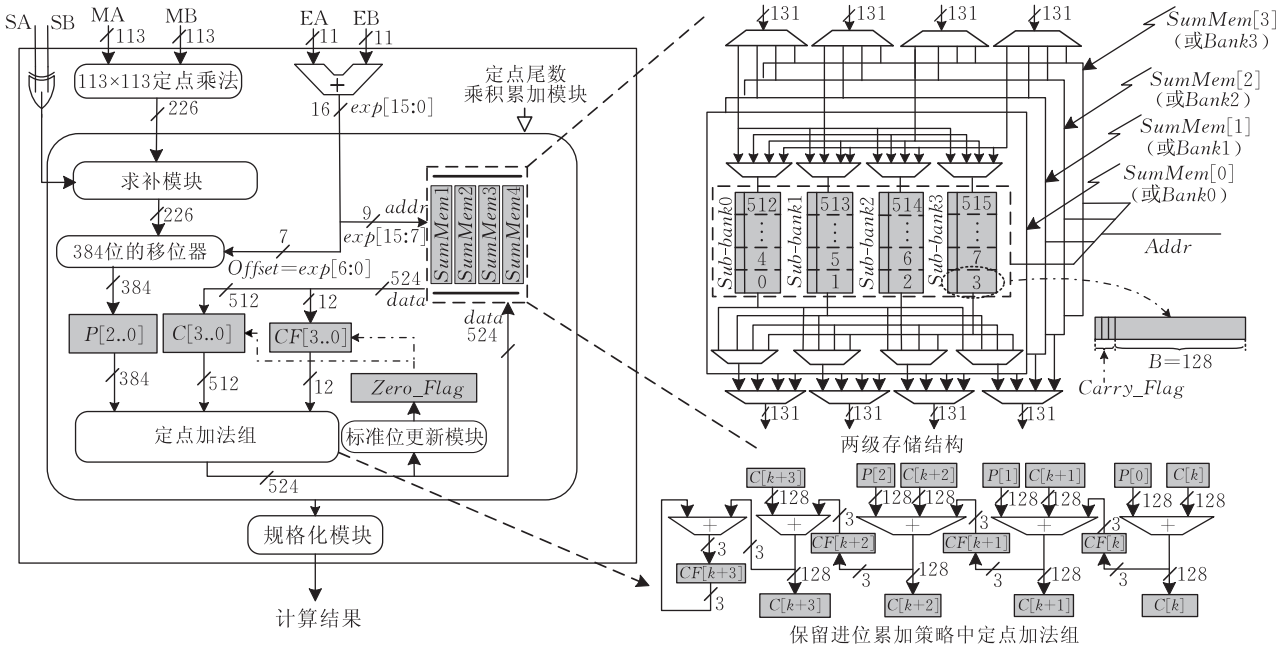


图 2　QPMAC 结构框图

3.2.1　两级存储策略

L 位定点累加和寄存器组织成一个两级存储结构. 如图 2 所示，第 1 级称为存储体(*SumMem* 或 *Bank*)，第 2 级称为子存储体(*Sub-bank*). 两级存储体的容量为 $L=|2e_1|+2r+2e_2=65\,756$ 位. 将 L 位存储体组织成一个多子存储体的结构，这能提供多个端口，快速访问与尾数乘积 *product* 对齐的数据.

我们为子存储体中每个项设置一个全 0 标志位(*ZF* 或 *Zero_Flag*)和一个进位标志位(*CF* 或 *Carry_Flag*). 当该项的值不为 0 时，置 *ZF* 为 1，否则置为 0. 全 0 标志位的作用：(1) 初始化存储体，即在累加操作前，设置存储体中对应的 *ZF* 为 0，而不需要对存储体进行初始化；(2) 规格化时快速确定 L 位 *sum* 中首 1 位置. 通过 *ZF* 可以快速定位首个非全 0 的字，这样将 L 位的首 1 查找转换为 B 位(子存储体的端口宽度)的首 1 查找. 进位标志位将在下面进行介绍.

我们根据 B 来组织两级存储结构. 通过选择合适的端口宽度以均衡加法长度、选择器数量、标志位的长度等硬件开销. 设：

B 为子存储体的端口宽度，为 2 的幂；

M 为定点尾数乘积 *product* 的位宽；

NS 为每个存储体中子存储体的个数，通常为 2 的幂，这样子存储体的访问地址为乘积指数 *exp* 的低位；

WC 为进位标志位的宽度，后面分析得 $WC=3$；

WA 为定点加法组中加法器宽度，为 $B+WC$；

NA 为定点加法组中 WA 位加法器的个数；

NM 为两级存储体结构中 B 位选择器的个数；

LF 为 *Zero_Flag* 标志位的长度，

则有：$NA=\lceil M/B\rceil+1$，$NS=2^{\lceil\log_2^{NA}\rceil}$，$NM=4\times(NS+NA)+NA$，$LF=\lceil L/B\rceil$.

表 2 可由上述公式推导出来，可以看出情况 2 ($B=128$)为较优的折中选择，即每个存储体由 4 个子存储体组成，子存储体的端口宽度为 128 位，每个周期能够访问 4 个连续的 128 位数据.

表 2　端口宽度和硬件复杂度选择方案

情况	B	NS	NA	WA	NM	LF
1	64	8	6	67	62	1028
2	128	4	4	131	36	514
3	256	4	3	259	31	257

3.2.2　保留进位累加策略

本文采用保留进位累加策略实现定点尾数乘积的累加(下面简称为乘积累加)，每次累加计算选择对应字的进位标志位参与计算，并根据计算结果更新进位标志位.

Algorithm: Product Accumulation based on Carry-Save Scheme
Input: $Product_i$, $exp_i[15:0]$(i=1,2,…,n)
Output: $C[515:0]$, $CF[515:0]$, $Zero\_Flag[515:0]$
Initialize: $ZF[515:0]=0$;
1. **for** $i=1$ **to** n **do**
2. $\{P_i[2],P_i[1],P_i[0]\}=Product_i \ll exp_i[6:0]$;
3. Set $k=exp_i[15:7]$. Load $C[k]\sim C[k+3]$ and $CF[k]\sim CF[k+3]$ from $SumMem$;
4. **if** $(ZF[k]==0)$ **then** $\{CF[k],C[k]\}=0$;
if $(ZF[k+1]==0)$ **then** $\{CF[k+1],C[k+1]\}=0$;
if $(ZF[k+2]==0)$ **then** $\{CF[k+2],C[k+2]\}=0$;
if $(ZF[k+3]==0)$ **then** $\{CF[k+3],C[k+3]\}=0$;
5. $\{CF[k],C[k]\}=C[k]+P_i[0]$;
$\{CF[k+1],C[k+1]\}=C[k+1]+P_i[1]+CF[k]$;
$\{CF[k+2],C[k+2]\}=C[k+2]+P_i[2]+CF[k+1]$;
$\{CF[k+3],C[k+3]\}=\{CF[k+3],C[k+3]\}+CF[k+2]$;
6. Store $C[k]\sim C[k+3]$ and $CF[k-1]\sim CF[k+3]$ to $SumMem$;
7. Updata $ZF[k]\sim ZF[k+3]$ according to $C[k]\sim C[k+3]$ and $CF[k-1]\sim CF[k+3]$;
8. **end for**

图 3 基于进位保留策略的定点尾数乘积的累加算法

如图 2 和如图 3 所示,基于保留进位策略的乘积累加过程如下:

步 1(RdC):对齐 *sum* 和 *product*,本文采用两级对齐策略:

步 1.1,乘积指数的高 9 位作为 *sum* 的访问地址(假定 $k=exp[15:7]$),读出与 *product* 对齐的连续 4 个 128 位的字 $C[k+3:k]$及相应的进位标志 $CF[k+3:k]$,同时根据符号位将 *product* 转换为补码表示.

步 1.2,然后根据乘积指数的低 7 位对 *product* 进行移位,使之与$C[k+3:k]$对齐,得到 $P[2:0]$;

步 2(CalC):使用 4 个 128 位的定点加法器完成定点尾数乘积的累加,使用 1 个 3 位的进位加法完成进位处理,如图 2 所示.

步 3(WrC):将累加结果 $C[k+3:k]$和更新后的进位标志位 $CF[k+3:k]$写回到 *sum* 的相应位置.

步 4(UpF):根据 $C[k+3:k]$和 $CF[k+3:k]$更新相应的全 0 标志.

定理 1. 采用上述保留进位累加策略,累加次数小于 2^{127} 次时,每个进位标志的数值不会超过 3.

证明. 假定 $W[i]=\{CF[i],C[i]\}$,采用归纳法证明第 j 次累加后,$W[i]<2^{129}+2\times j$,其中 $0\leq i\leq 515$.

(1)初始化时,$W[i]=0$,$0\leq i\leq 515$.

(2)第 $j+1$ 次累加时,假定 $W[k]$、$W[k+1]$、$W[k+2]$、$W[k+3]$参与累加计算,根据归纳假设有:

$W[k]=P[0]+C[k]<2^{129}+2\times(j+1)$;

$W[k+1]=P[1]+C[k+1]+CF[k]<2^{129}+2\times(j+1)$;

$W[k+2]=P[2]+C[k+2]+CF[k+1]<2^{129}+2\times(j+1)$;

$W[k+3]=W[k+3]+CF[k+2]<2^{129}+2\times(j+1)$.

其余字和对应的进位标志位保持不变,这可保证第 $j+1$ 次累加结果满足 $W[i]<2^{129}+2\times(j+1)$,$0\leq i\leq 515$. 若进位标志数值超过 3,则 $2^{129}+2\times(j+1)>3\times 2^{128}$,即 $j>2^{127}-1$. 证毕.

因此每个进位标志位仅用三位来表示,高位表示进位符号、低两位表示进位数值.

相对于快速进位处理方法[2,17-18],这种保留进位累加策略的优势在于:

(1)将文献[2]中的 384 位加法转化为 4 个并行执行的三输入 128 位加法,减少延时. 同样采用先行进位设计方法,将三输入 128 位加法分成 4 段,13 个 32 位的加法操作并发执行,最后根据各段的进位选择最终结果. 这种设计方法将三输入 128 位加法延时从 6.48 ns 降低到 4.35 ns,且小于 384 位加法延时(6.8 ns)[2].

(2)简化进位处理逻辑. 快速进位方法中传递因子和终止因子处理过程:首先,根据乘积指数 *exp* 对 *all_one_flag* 标志位[2]进行左移操作;然后,进行数 1 操作确定移位后的标志位寄存器中最低的 0 位,对应的字为终止因子;最后,更新传递因子和终止因子,并进行右移操作,完成 *all_one_flag* 标志位更新操作. 从 DD 类型扩展到四精度时,*all_one_flag* 标志位宽度由 36 位增大到 516 位,导致快速进位过程中的移位操作和数 1 操作的延时达到 11.0 ns 和 10.1 ns,导致 HPMAC 单元的频率降低. 保留进位累加策略采用进位标志位替换传递因子、终止因子的确定及进位与终止因子加法逻辑,从而提高 QPMAC 单元的频率和性能.

3.2.3 累加和划分策略

乘积累加需要进行累加和寄存器访问、定点加法计算、标志位更新等操作. 整个过程延时较大,需要使用流水线技术来提高运行频率. 根据保留进位累加策略,将乘积累加模块设计成四级流水线结构,分别为 RdC、CalC、WrC 和 UpF.

同时,本文提出累加和划分策略来实现全流水乘积累加模块,使 QPMAC 获得流水吞吐率. 该策略将累加和 *sum* 分为 4 个部分,分别存储到存储体 $SumMem[0]\sim SumMem[3]$中. 多存储体流水执行的时空图如图 4 所示,每个存储体 4 个周期完成一次乘积累加操作,每个时钟周期各个存储体处于不同流水阶段. 假如 *product* 在周期 j 进入乘积累加模块,则将其累加到存储体 $SumMem[j\%4]$中,其中"%4"表示模 4 操作.

总之,累加和两级存储策略中,第 1 级由 4 个存

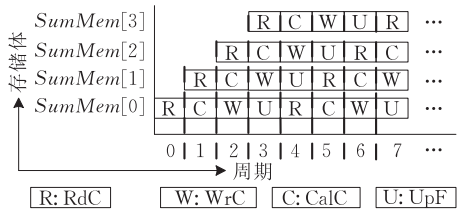


图 4　多存储体流水执行时空图

储体(*SumMem*[0～3])组成，这使 QPMAC 单元能够获得流水吞吐率；第 2 级由 4 个子存储体(*Sub_bank*[0～3])组成一个存储体，这能够提供多个访问端口，每个周期可以访问多个数据，减少数据访问周期.

3.2.4　QPMAC 实现四精度浮点基本操作

四精度浮点加法、乘法及乘加融合(FMA)操作可以视为内积运算的特殊情况. 浮点加法可视为向量长度为 2 的内积，且向量 $\boldsymbol{B}$ 的元素均为 1；浮点乘法可视为向量长度为 1 的内积；FMA 操作($r=a\times b+c$)，可视为向量长度为 2 的内积，且向量 $\boldsymbol{A}=\{a,c\}$，向量 $\boldsymbol{B}=\{b,1\}$.

4　基于 QPMAC 的应用

本节以 Doolittle LU 分解和 MGS-QR 分解应用为例来说明 QPMAC 的性能及高精度科学计算 FPGA 加速器的设计方法. 加速器结构如图 5 所示.

4.1　基于 QPMAC 的 LU 分解

LU 分解是 Linpack 测试基准中的核心算法. 如下所示，Doolittle LU 分解方法[13]由向量内积和除法操作组成，它将非奇异矩阵 $\boldsymbol{A}$ 分解为下三角矩阵 $\boldsymbol{L}$ 和上三角解矩阵 $\boldsymbol{U}$，计算复杂度为 $O(2n^3/3)$.

for $k=1$ **to** n
  for $j=k$ **to** n
    $u_{k,j}=a_{k,j}-\sum_{i=1}^{k-1} l_{k,i}u_{i,j}$　▶(1) Cal element of $\boldsymbol{U}$
  end for
  for $i=k+1$ **to** n
    $l_{i,k}=(a_{i,k}-\sum_{j=1}^{k-1} l_{i,j}u_{j,k})/u_{k,k}$　▶(2) Cal element of $\boldsymbol{L}$
  end for
end for

图 6 描述了基于 QPMAC 细粒度流水 LU 分解算法. 由两个并行算法组成，分别为 Master 算法和 Slave 算法，它们采用基于消息传递接口(MPI)的 SPMD 模式来描述，其中，N 表示矩阵 $\boldsymbol{A}$ 的规模，P 表示执行 Slave 算法的处理单元(PE)个数.

如图 5(a)所示，LU 分解加速器主要由接口控制模块、主 PE 和一个计算 PE 阵列组成. 其中，接口控制模块通过 PCIE 通路与主机进行数据和命令交互，同时还控制两个 DDR2 存储条的访问；主 PE 执行 Master 算法，计算 PE 执行 Slave 算法，主 PE 和计算 PE 阵列共同完成 LU 分解的计算任务.

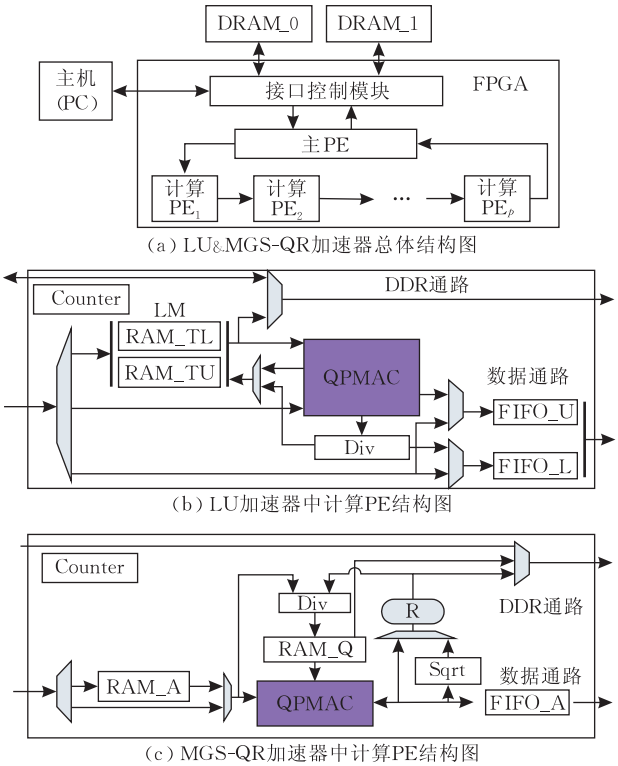


图 5　LU 分解和 MGS-QR 分解加速器

主 PE 和计算 PE 阵列均运行 N/P 次. 每次运行主 PE 从 DDR2 存储器中读取数据，发送到计算 PE_1，执行发送原语 5 和 6；当所有计算 PE 完成计算后，主 PE 执行写回原语 9 和 10，将 $\boldsymbol{L}$ 矩阵的 P 列和 $\boldsymbol{U}$ 矩阵的 P 行写回到 DDR2 存储器.

计算 PE 的结构如图 5(b)所示，主要包括两个 RAM、两个 FIFO、一个 QPMAC 单元、一个四精度浮点除法模块和控制逻辑. 两个 RAM(RAM_TL 和 RAM_TU)存储计算结果中 $\boldsymbol{L}$ 矩阵的一列和 $\boldsymbol{U}$ 矩阵的一行；两个 FIFO(FIFO_L 和 FIFO_U)用于缓冲计算 PE 之间的数据；四精度浮点除法模块采用基 4-SRT 除法算法[19]实现.

QPMAC 单元能够独立计算原语 *Cal_U*，由 QPMAC 和除法模块共同计算原语 *Cal_L*. 注意到：图 6 算法 A 中原语 *Cal_L*(Line9)执行之前，向量 ***TL*** 已经存储在 RAM_TL 中，因此，*Cal_L* 的计算是由向量 ***TD*** 的数据所驱动，接收原语(Line8)、*Cal_L* 原语(Line9)和发送原语(Line11)能够并行执行. 同理，接收原语(Line12)、*Cal_U* 原语(Line13)和发送原语(Line15)能够并行执行. 这种计算与数据通信相重叠的方式可以有效掩盖数据传

递延时,数据以流水方式在 PE 之间传递,这最大限度发挥了 QPMAC 单元的流水吞吐率优势,达到每个周期完成一次四精度乘累加操作.

Algorithm A: Pipelined LU decomposition algorithm

Master algorithm:

```
1.  if pid=0 then
2.   for k=1 to N by P
3.    M=SetLength(k); Set k'=k+pid;
4.    do in parallel
5.     Send(1, A[k:N, 1:(k+P)]);
6.     Send(1, A[1:(k+P), k:N]);
7.    end do
8.    for pid=1 to P
9.     Store(A[k':N, (k'-1)], pid, TL[k':N]);
10.    Store(A[(k'-1), (k'-1):N], pid, TU[(k'-1):N]);
11.   end for
12.  end for
13. end if
▷Store(X, pid, Y): store elements of an array Y of PE numbered
  pid into X
▷Send(pid, X): send elements of an array X to PE numbered pid;
▷Rcv(pid, X): receive elements of an array X from PE numbered pid;
▷Cal_U(X[1:n-1], Y[1:n], u): calculate elements of U
▷Cal_L(X[1:n-1], Y[1:n], a, l): calculate elements of L
```

Slave algorithm:

```
1.  if 1≤pid≤P then
2.    while(TRUE)
3.      WaitSetLength(M); Set M'=M+pid-1;
4.      Rcv(pid-1, TL[1:M+P-1]);
5.  Parallel┌Rcv(pid-1, TU[1:M+P-1]);
6.    Do    └Cal_U(TL[1:M'-1], TU[1:M'], TU[M'])
7.      for i=M+pid to N
8.          ┌Rcv(pid-1, TD[1:M+P-1]);
9.   Parallel Cal_L(TL[1:M'-1], TD[1:M'], TU[M'], TL[i]);
10.    Do    TD[M']=TL[i];
11.         └Send(pid+1, TD[1:M+P-1]);
12.         ┌Rcv(pid-1, TD[1:M+P-1]);
13.  Parallel Cal_U(TU[1:M'-1], TD[1:M'], TU[i]);
14.    Do    TD[M']=TU[i];
15.         └Send(pid+1, TD[1:M+P-1]);
16.     end for
17.   end while
18. end if
```

Algorithm B: Pipelined MGS-QR decomposition algorithm

Master algorithm:

```
1.  if pid=0 then
2.   for k=1 to N by P
3.    M=Setlength(k);
3.    Send(1, A[k:N, 1:N]);
4.    Store(A[k+p:N, 1:N], P,FIFO_A);
5.    for pid=1 to P
6.     Store(R[(k+pid),(k+pid):N],pid,TR[(k+pid):N]);
7.     Store(Q[(k+pid), 1:N], pid, TQ[1:N]);
8.    end for
9.   end for
10. end if
▷Cal_R(X[1:n], Y[1:n], u): execute the dot product u of vector
  X and Y
▷Cal_Q(X[1:n],R,Y[1:n]): execute the operation Y[i]=X[i]/R
▷Cal_A(X[1:n],R,Y[1:n]): execute the operation X[i]=X[i]-
  R*Y[i], where i=1,2,…,n
```

Slave algorithm:

```
1.  if 1≤pid≤P then
2.    while(TRUE)
3.      WaitSetLength(M);
4.  Parallel┌Rcv(pid-1, TA[1:N]);
5.    Do    └Cal_R(TA[1:N],TA[1:N],Temp_R);
6.          TR[M+pid]=sqrt(Temp_R);
7.          Cal_Q(TA[1:N],TR[M+pid],TQ[1:N]);
8.          for i=M+pid+1 to N
9.        Parallel┌Rcv(pid-1, TA[1:N]);
10.         Do    └Cal_R(TA[1:N],TA[1:N],TR[i]);
11.       Parallel┌Cal_A(TA[1:N],TD[i],TQ[1:N]);
12.         Do    └Send(pid+1, TA[1:N]);
13.         end for
14.     end while
15. end if
```

图 6 细粒度流水 LU 分解算法和 MGS-QR 分解算法

4.2 基于 QPMAC 的 MGS-QR 分解

MGS-QR 分解算法[13]如下所示,该算法将非奇异矩阵 $\boldsymbol{A}$ 分解为正交矩阵 $\boldsymbol{Q}$ 和上三角矩阵 $\boldsymbol{R}$,计算复杂度为 $O(n^3)$.

```
for k=1 to N
  R[k,k]=sqrt(Σ_{i=1…N} A[k,i]·A[i,k]);
  Q[k,*]=A[k,*]/R[k,k];
  for j=k+1 to N
    R[k,j]=Σ_{i=1…N} Q[k,i]·A[i,j];
    A[j,*]=A[j,*]-R[k,j]·Q[k,*];
  end for          ▲ *=1,2,…,N
end for
```

基于 QPMAC 流水 MGS-QR 分解算法及加速器结构与 LU 分解算法相似,如图 5、6 所示. Master 算法的主要区别是从DDR2存储器中读出的数据和传递到计算 PE 中的数据不同. LU 分解中,每个计算 PE 是对原始矩阵的数据进行操作;而 MGS-QR 分解中,每个计算 PE 需要更新整个矩阵,下一个计算 PE 的数据是当前计算 PE 的更新结果,Master 算法和 Slave 算法每执行一次,都需要将最后计算 PE 的更新矩阵写回到 DDR2 存储器中.

如图 5(c)所示,MGS-QR 分解加速器中的计算 PE 主要由两个 RAM、一个 FIFO、一个 QPMAC 单元、一个四精度浮点除法模块和一个四精度浮点开方模块. RAM_A 用于存储来自上一个 PE 的更新

数据，RAM_Q 用于存储计算结果中 $\boldsymbol{Q}$ 矩阵的一行；FIFO_A 用于缓冲计算 PE 之间的数据；四精度开方模块采用 Non-Restoring 开方算法[20]实现.

与 LU 分解相似，QPMAC 单元能够独立计算原语 *Cal_R* 和 *Cal_A*，采用数据传递和计算重叠的方法来隐藏通信延时，提高 QPMAC 单元的利用率. 但与 LU 分解不同的是，图 6 算法 B 中 Slave 算法的第 11、12 行必须在第 9、10 行完成后才能执行，这就使得数据接收、计算和发送不能并行执行. 只能通过数据接收与计算重叠和数据发送与计算重叠方式来隐藏数据通信延时.

5　实验结果

我们在 FPGA 开发板上验证上述设计. 开发板上包含一块 FPGA 芯片、2×2GB DDR2 存储条. DDR2 控制器的运行频率为 200 MHz，峰值带宽可达 6.4 GB/s. FPGA 芯片为 Xilinx Virtex5 XC5VLX330-1FF1760，它包含 207 360 个 LUT、192 个 DSP48E、10 368Kb 片内存储器. 主机与 FPGA 通过 PCIE x8 通道进行数据和命令的交互，带宽为 900 MB/s. 我们使用性能较高的任意精度函数库 MPFR 函数库[6]（MPFR 3.0.0，精度设为 113 位）和针对 Intel 处理器进行性能优化的 Intel Fortran 函数库（简称 Intel 函数库）[5]，来对比性能和结果精度. 软件平台主要包括四核处理器（2.33 GHz Intel Core2 Quad Q8200）、4 GB DDR3存储条. FPGA 平台的运行时间包含从主机向开发板发送初始数据和主机从开发板接收结果数据的时间.

5.1　FPGA 资源

表 3 描述了基本运算部件及不同 PE 个数的 LU 分解和 MGS-QR 分解加速器的综合结果.

表 3　FPGA 资源使用表

	规模	Slice Reg	Slice LUT	Block RAM	DSP48E	Freq/MHz	Power/W
基本运算部件	QPMAC	14000(6%)	27028(13%)	0	35(18%)	192.678	—
	DIV	855(0%)	1073(0%)	0	0	185.632	—
	SQRT	615(0%)	638(0%)	0	0	212.736	—
LU 分解加速器	2PE	39954(19%)	77504(37%)	76(26%)	70(36%)	167.870	18.50
	4PE	68957(33%)	143087(68%)	134(46%)	140(72%)	160.810	21.93
MGS-QR 分解加速器	2PE	39223(18%)	75242(36%)	76(26%)	70(36%)	184.264	17.04
	4PE	67808(32%)	142635(68%)	134(46%)	140(72%)	160.308	19.96

从表 3 中可以看出，DSP48E 资源已经成为四精度科学计算加速器设计的瓶颈. 因为 DSP48E 用于完成定点尾数乘法，它的需求量是以尾数宽度的平方速度增长. 在 QD HPMAC(256 位)中 DSP48E 的消耗达到了 50%[2].

在全流水 QPMAC 设计中，使用 FPGA 片内分布式 RAM 实现两级存储体结构中的子存储体. 这使 FPGA 的布局布线更加灵活，克服 FPGA 片内块 RAM 位置确定，结构固定的不足，同时这也能提高 FPGA 片内存储器资源的利用率. 但是，分布式 RAM 占用 FPGA 的逻辑资源. 因此，QPMAC 中 Slice LUT 消耗达到 13%.

由于在 QPMAC 设计中采用保留进位累加策略来克服数据范围扩大带来的挑战，综合频率相对于 DD HPMAC(170.7 MHz)[2]提高了 12%. LU 和 MGS-QR 分解加速器的综合频率都超过 160 MHz. 而且，计算阵列规模的扩大并没有导致加速器频率出现明显下降，这表明 LU 和 MGS-QR 分解加速器结构具有良好的可扩展. 对于 4PE 的 LU 或 MGS-QR 分解加速器来说，FPGA 的峰值性能达到 1280M FLOPS(128-bit Floating-point Operations Per Second). 下面对于向量内积、LU 及 MGS-QR 分解应用，FPGA 平台均运行于 133 MHz.

表 3 中的功耗是通过 ISE11.3 中的 XPower Analyzer 工具估计得到的，其中 20%～30%的功耗用于 IO buffer. 4PE 的 LU 分解加速器的功耗为 21.93 W，它仅为 4 核处理器 Intel Core2 Quad Q8200 功耗(95 W)的 1/4.

5.2　向量内积应用的精度和性能

图 7 比较了 FPGA 平台和软件平台实现不同长度向量内积运算的精度. 在文献[2]图 14(d)中定义的数据集 *Set*1～*Set*6 上测试内积运算，选取随机

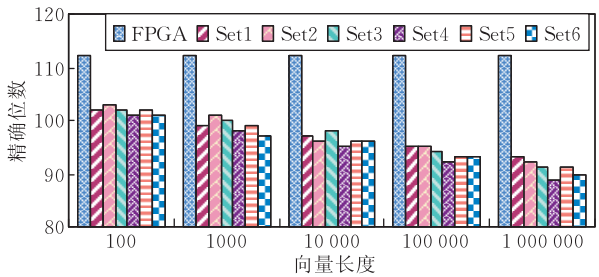


图 7　向量内积的精度比较

测试中最坏情况来说明.从图 7 可以看出,软件平台的内积结果精度会随着向量长度的增加而降低.而 QPMAC 计算过程中使用精确的定点运算,不会产生误差,因此能够获得精确的结果.两种方法的精度差别达到 12 位.

图 8 比较了软件平台(MPFR、Intel 函数库)和 FPGA 平台实现向量内积的性能.MPFR 和 Intel 函数库的吞吐率分别为 6M FLOPS 和 21M FLOPS,而 QPMAC 单元采用全流水设计,能够取得流水吞吐率,假定初始数据全部存储于 DDR2 存储条上,QPMAC 单元以 133 MHz 运行能取得 266M FLOPS 的性能,加速比分别达到 45.9 和 12.8.

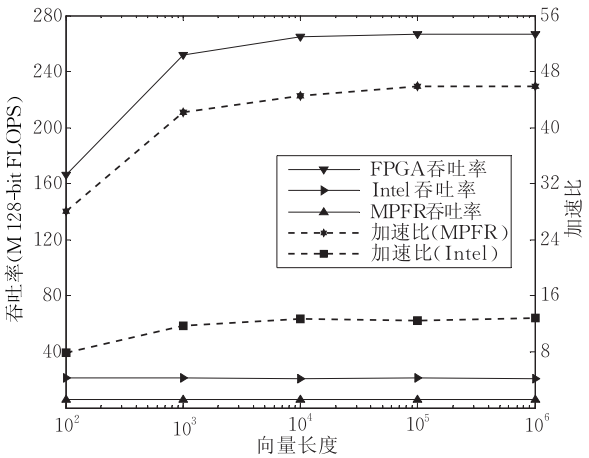


图 8 向量内积的性能比较

我们通过 Intel VTune 性能分析工具来分析 CPU 和 FPGA 平台的性能.如表 4 所示,任意精度函数库 MPFR 实现 1M 的向量内积需要 1126M 条指令,这是因为 MPFR 函数库中,任意精度浮点操作使用整数操作模拟实现,所需的指令数较多,而且还有部分指令用于函数调用等.Intel 函数库是针对 IEEE 标准的四精度浮点算术进行专门设计,相对 MPFR 来说,其指令数可以减少一半以上,而且 Intel函数库针对 Intel 处理器的特点(如 SSE)进行性能优化,其 CPI 更低,因此,Intel 函数库的性能比 MPFR 高 2.6 倍.FPGA 平台能够针对不同的计算精度设计相应的逻辑单元,所需指令数较少,而且 QPMAC单元采用流水设计,因此,FPGA 平台能够以 133 MHz 的运行频率取得 12 倍以上的加速比.

表4 不同平台下长度为 1M 的向量内积的性能分析

MPFR(113 bits)			Intel Fortran			FPGA		
N. I.	*CPI*	*T*/ms	*N. I.*	*CPI*	*T*/ms	*N. I.*	*T*/ms	*S*/ms
1126M	0.7	344	503M	0.44	95.8	2M	7.5	12

注:表中:*N. I.* 表示指令数,*CPI* 表示 Clock Per Instruction,*T* 表示时间,*S* 表示相对于 Intel 函数库的加速比.

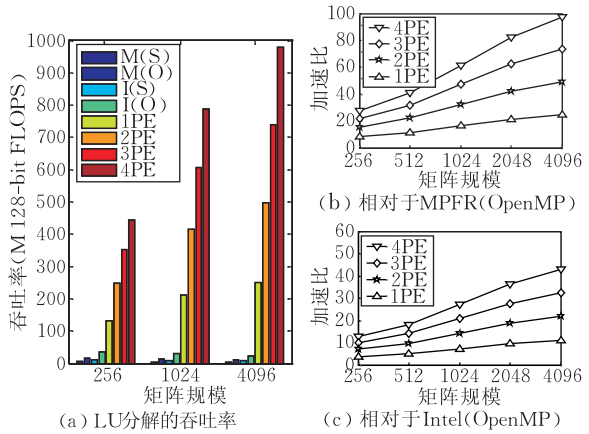
5.3 LU 分解应用的精度和性能

如表 5 所示,基于 QPMAC 的 LU 分解的计算精度比相同精度的软件实现更精确.这是因为软件实现向量长度 n 的内积时,需要引入 $2n-1$ 次舍入误差,而 QPMAC 单元仅在规格化时才进行一次舍入操作.对于规模为 4096 的 LU 分解,结果精度提高了 5.4 位.

表 5 LU 分解和 MGS-QR 分解精度比较

	平台	精确位数				
		256	512	1024	2048	4096
LU	FPGA	100.1	98.8	97.5	95.7	94.1
	CPU	97.3	95.1	93.1	90.9	88.7
MGS-QR	FPGA	103.6	102.3	100.9	98.8	97.1
	CPU	100.8	99.6	97.1	93.7	91.2

图 9 比较了 FPGA 平台和软件平台上不同规模 LU 分解的性能.在 Intel 四核处理器上实现串行 LU 分解算法和基于 OpenMP 的并行 LU 分解算法.对于 MPFR 和 Intel 函数库,并行算法的执行速度分别为串行算法的 2.8 倍和 2.65 倍.从图 9(a)可知,对于给定矩阵规模,LU 分解加速器的吞吐率随着 PE 个数成线性增长,这是因为随着 PE 个数的增加,Master 和 Slaver 算法的运行次数线性减少,而每次运行的时间开销基本保持不变.对于确定 PE 个数,LU 分解加速器的吞吐率随着矩阵规模的增大略有提高,这是因为计算 PE 阵列的启动和流水排空开销固定不变,而数据以流水方式在计算 PE 之间传递,随着矩阵规模的增大,启动和流水排空开销在总时间的比重减小.当矩阵规模为 4096 时,FPGA 平台上 4PE 的 LU 分解加速器的吞吐率接近 1000M FLOPS,而 MPFR 和 Intel 函数库的吞吐率分别为 10.1M FLOPS 和 22.7M FLOPS,加速比可以达到 97 倍和 43 倍.



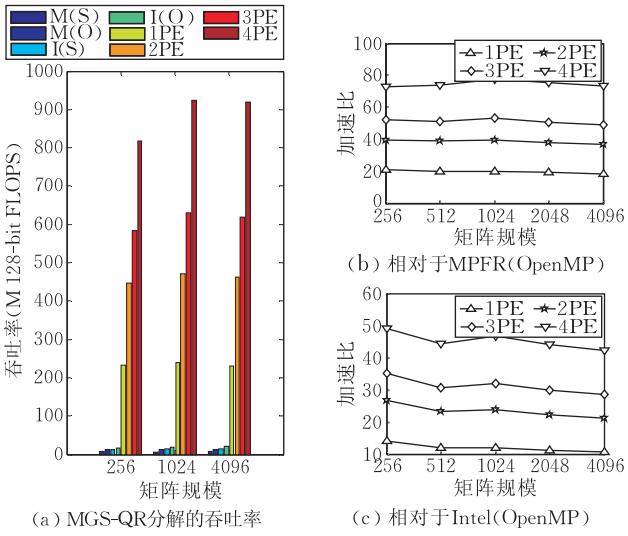
(a) LU分解的吞吐率 (b) 相对于MPFR(OpenMP) (c) 相对于Intel(OpenMP)

图 9 LU 分解的性能比较,其中图(a)中 M、I、S、O 分别表示 MPFR、Intel Fortran、串行算法、基于 OpenMP 的并行算法,柱状图表示不同实现方式的吞吐率,从左至右依次为:串行 MPER、并行 MPER、串行 Inter Fortran、并行 Inter Fortran 及集成(1PE、2PE、3PE、4PE)的 FPGA 加速器

5.4　MGS-QR 分解应用的精度和性能

与 LU 分解应用相似，基于 QPMAC 单元的 MGS-QR 分解的计算精度比相同精度的软件实现更精确. 如表 5 所示，对于矩阵规模为 4096 的 MGS-QR 分解，结果精度提高了 5.9 位.

图 10 比较了不同平台不同规模 MGS-QR 分解的性能. 在 Intel 四核处理器上，基于 OpenMP 的 MGS-QR 并行分解算法的执行速度约为串行算法的 1.9 倍和 1.8 倍. 与 LU 分解相似，对于给定矩阵规模，MGS-QR 分解加速器的吞吐率随 PE 个数线性增长. 但是对于确定 PE 个数，MGS-QR 分解加速器的吞吐率不再随着矩阵规模增大而提高，这是因为 MGS-QR 分解算法中矩阵更新过程的数据接收、计算和发送不能同时执行，数据不能以完全流水的方式在计算 PE 阵列中流动，这导致不同矩阵规模，计算 PE 阵列的启动和流水排空开销在总时间的比重保持不变. 对应矩阵规模为 4096 的 MGS-QR 分解，4PE FPGA 实现的吞吐率达到 918M FLOPS，而 MPFR 和 Intel 函数库的吞吐率分别为 12.5M 和 21.7M FLOPS，加速比达到 73 倍和 42 倍.



(a) MGS-QR分解的吞吐率　(b) 相对于MPFR(OpenMP)　(c) 相对于Intel(OpenMP)

图 10　MGS-QR 分解的性能比较，子图中 M、I、S、O 的定义和柱状图的定义与图 9 相同

5.5　PCI-E 性能分析

由于主机与 FPGA 通过 PCIE x8 通道进行数据交互，FPGA 平台的性能受到 PCIE 带宽的限制. 在向量内积应用中，QPMAC 的计算性能为 266.6M FLOPS，匹配的 IO 带宽为 4.3 GB/s，而实际通信带宽仅为 900 MB/s，这导致 FPGA 平台的计算性能下降 4.8 倍，相对于 Intel 函数库的加速比仅为 2.7. 在 LU 和 MGS-QR 分解应用中，由于计算复杂度与通信复杂度的比为 $O(n)$，通信开销在总时间中的比重是随着矩阵规模增大而减小，如图 11 所示. 因此，使用 FPGA 平台加速高精度科学应用时，需要尽量复用数据，提高计算复杂度与通信复杂度的比值，减少 IO 带宽的需求，从而提高加速性能.

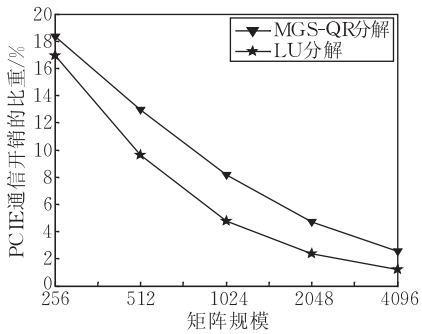


图 11　LU 和 MGS-QR 分解中 PCIE 通信开销相对于 4PE FPGA 实现总时间的比重

6　结　论

本文定制了一个四精度浮点乘累加单元(QPMAC)，采用精确定点乘法和加法操作代替浮点操作，仅在最后规格化时进行一次舍入操作. 这样能够获得精确的向量内积结果，避免减法操作的“巨量相消”现象. 在 QPMAC 设计中，提出两级存储策略来存储累加和的数据，同时引入保留进位累加策略和累加和划分策略，来最小化关键路径长度，简化进位处理，保证流水吞吐率. 最后，为了验证 QPMAC 性能，在 XC5VLX330 芯片上设计一个 LU 分解和 MGS-QR 分解加速器原型，集成 4 个 QPMAC 单元，与 Intel 四核处理器相比获得 42 到 97 倍的性能提升.

参　考　文　献

[1] Kogge P et al. Exascale computing study: Technology challenges in achieving exascale systems. Government PROcurement, USA: Technology Report: DARPA-2008-13, 2008

[2] Dou Yong, Lei Yuan-Wu, Wu Gui-Ming, Guo Song. Fpga accelerating double/quad-double high precision floating-point applications for exascale computing//Proceedings of the 24th International Conference on Supercomputing, Tsukuba, Japan, 2010: 325-336

[3] Bailey D H. High-precision floating-point arithmetic in scientific computation. Computing in Science and Engineering, 2005, 7(3): 54-61

[4] IEEE Standard for binary floating point arithmetic ANSI/IEEE standard 754-2008. The Institute of Electrical and Electronic Engineers, 2008

[5] Intel Compilers and Libraries [Online 2011]. http://soft-

ware. intel. com/en-us/articles/intel-compilers/
[6] Fousse L, Hanrot G, Lefevre V et al. Mpfr: A multiple-precision binary floating-point library with correct rounding. ACM Transactions on Mathematical Software (TOMS), 2007, 33(2): 1-14
[7] Hida Y, Li X S, Bailey D H. Quad-double arithmetic: Algorithms, implementation, and application. Lawrence Berkeley National Laboratory, Berkeley, CA, Report LBL-46996, 2000
[8] Fujimoto J, Ishikawa T, Perret-Gallix D. High precision numerical computations: A case for a happy design. ACPP IRG note, ACPP-N-1: KEK-CP-164, 2005
[9] Schwarz E M, Smth R M, Krygowski C A. The s/390 g5 floating point unit supporting hex and binary architectures// Proceedings of the 14th IEEE Symposium on Computer Arithmetic. Adelaide, Australia, 1999: 836-841
[10] Akkas A, Schulte M J. Dual-mode floating-point multiplier architectures with parallel operations. Journal of Systems Architecture: The EUROMICRO, 2006, 52(10): 549-562
[11] Rump S M. Algorithms for verified inclusions-theory and practice//Reliability in Computing. New York: Academic Press, 1988
[12] Hasegawa H. Utilizing the quadruple-precision floating-point arithmetic operation for the Krylov Subspace methods//Proceedings of the SIAM Conference on Applied Linear Algebra. Williamsburg, VA, 2003
[13] Higham N J. Accuracy and Stability of Numerical Algorithms. 2nd edition. Society for Industrial and Applied Mathematics Philadelphia, PA, USA, 2002
[14] Rump S M, Ogita T, Oishi S. Accurate foating-point summation part i: Faithful rounding. SIAM Journal on Scientic Computing (SISC), 2008, 31(1): 189-224
[15] Kulisch U. The fifth floating-point operation for top-performance computers. Universitat Karlsruhe, Karlsruhe, Germany: Technology Repor D-76128, 1997
[16] He C, Qin G et al. Group-alignment based accurate floating-point summation on FPGAs//Proceedings of the ERSA 2006. Las Vegas, Nevada, USA, 2006: 136-142
[17] Kulisch U. Computer arithmetic and validity: Theory, implementation, and applications. New York, Berlin: De Gruyter, 2008
[18] Knofel A. A fast hardware units for the computation of accurate dot products//Proceedings of the Arith10. Grenoble, France, 1991: 70-74
[19] Parhi K K, Srinivas H R. A fast radix-4 division algorithm and its architecture. IEEE Transactions on Computers, 1995, 44(6): 826-831
[20] Li Ya-Min, Chu Wan-Ming. Parallel-array implementations of a non-restoring square root algorithm//Proceedings of the IEEE International Conference on Computer Design. Austin, Texas, USA, 1997: 690-695

LEI Yuan-Wu, born in 1982, Ph. D. candidate. His research interests include high performance computer architecture.

DOU Yong, born in 1966, Ph. D., professor, Ph. D. supervisor. His research interests include high performance computer architecture, high performance embedded microprocessor and reconfigurable computing.

GUO Song, born in 1985, Ph. D. candidate. His research interests include high performance computer architecture.

Background

Scientific computing is becoming the third mode of scientific discovery. Most scientific computation applications operate on 64-bit floating-point arithmetic, in which rounding errors are an unavoidable consequence. The accumulation of rounding errors in some large-scale applications leads to inaccurate, unreliable and even wrong results. Therefore, a rapidly expanding body of applications, ranging from mathematical computations to large-scale physical simulations, require high-precision arithmetic. Currently, most of high-precision arithmetic is accomplished by software emulation, such as MPFR (Multiple Precision Floating-Point Reliable), GMP (GNU Multiple-precision arithmetic), NTL, QD library. However, the performance of high-precision arithmetic, using software, drops by at least one order of magnitude, compared to 64-bit floating-point. Some hardware designs attempt to overcome the speed limitations of software approach, such as CADAC, VPIAP. Recently, the use of FPGA-based accelerators has become a promising approach for speed up scientific applications.

We analyzed the accuracy loss in floating-point arithmetic and large-scale LU decomposition, and implemented a Double-Double and Quad-Double HPMAC unit in the paper appeared in the 24th International Conference on Supercomputing (ICS' 2010) [FPGA Accelerating Double/Quad-Double High Precision Floating-Point applications for ExaScale Computing]. This paper is an extended version and improvement of that paper. First, we present a full-pipelined multiplication and accumulation on FPGA for IEEE 754-2008 quadruple-precision floating-point arithmetic. We employ specific schemes to improve the precision and the performance of applications. The experimental evaluation demonstrates that the performance of FPGA approach outperforms general-purpose computers with a speedup of more than 42X, however the power consumption is only about 25% of the latter. So we believe the application-specific scheme implemented in high-precision scientific computation applications provides significant capability and flexibility over the general-purpose schemes. This work is sponsored by the National High Technology Research and Development Program (2008AA01A201) and the National Natural Science Foundation of China (60833004 and 61125201).